

AD-A165 328

CHARACTERIZATION OF AN ADA SOFTWARE DEVELOPMENT(U)
MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE
V R BASILI ET AL SEP 85 N00014-82-K-0225

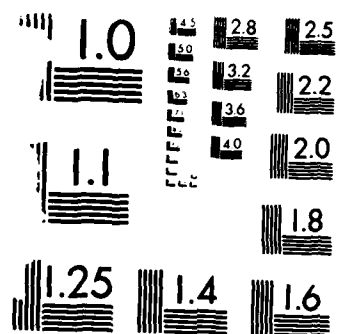
1/1

UNCLASSIFIED

F/G 5/9

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Characterization of an Ada Software Development

N00014-82-K-0225

Victor R. Basili, Elizabeth E. Katz, Nora Monina Panlilio-Yap, Connie Loggia Ramsey, and Shih Chang

University of Maryland

AD-A165 320

The University of Maryland and General Electric monitored an industrial software development project in Ada at GE. The findings will aid in the training, tool support, and methodology for future Ada projects.

Accession for

NTIS GRA&I

DTIC TAB

Unannounced

Justification

By

For

Project

Task

Subtask

Phase

Activity

Product

Document

Page

Number

Date

Author

Title

Subject

Ada is already required for use in embedded software of the United States Department of Defense. In time, more government agencies and private-sector companies will be using Ada, yet it is not without its problems. Though praised for being comprehensive, Ada has also been criticized for being cumbersome. Its effective use by people with varied backgrounds will depend on training, available tools, and the application of both to the project at hand.

This article examines the use of Ada in a software project developed by the General Electric Company. The project was monitored by the University of Maryland and GE to identify areas of success and difficulty in learning and using Ada as both a design and a coding language. Since production-quality Ada translators were not readily available, the study focused on training and early software development. We focus on the use and effect of Ada on this project, which was conducted primarily in 1982. Our study also presents the major factors to consider before using Ada in software development, particularly when training in Ada is necessary. Although many of our conclusions may seem obvious now, they were unexpected when this project began.

Our study attempts to meet several goals. The first focuses on characterization of the effort, the changes, and the errors of the project. The second considers how Ada was used on the project. The third concerns evaluation of the data collection and validation process, while the fourth concentrates on the development of measures for the Ada Programming Support Environment.

The development of metrics is an on-going project discussed by Basili and Katz¹ and Gannon et al.,² so Ada-specific metrics will not be discussed here except as they apply to this project.

Data sources and validation

From the start of the project, data was collected from a variety of sources. Change data, in particular, is often gathered only after the code has been compiled and entered into a "system" for use by other project members. In this project, however, all changes made after the text was on-line were included. Therefore, comparisons with data of other projects may be misleading. However, some of the data will be presented with compiler-detectable faults eliminated so comparisons can be made with the early

A-1

September 1985

AD-A165 320

53

86

3

109

Figure 1. Change request form.

Data sources. The programmers were asked to complete various forms. Each week, a component status report form indicated how each programmer's effort was distributed by component involved and by phase of development. Each time a need for change arose, a change-request form was completed. If the change was an error

In addition, a copy of each design and code version of every module was kept. The source code measures were

Data validation. Data collected on forms is difficult to validate,⁴ but valid data is necessary for valid results; therefore, the completed forms were screened for inconsistencies. The programmers could then clarify discrepancies before data was analyzed. Screening should be done as soon as possible in future studies. If the forms could be completed on-line, checked for inconsistencies, and automatically entered into a database, validation could proceed more quickly. We suggest that future data collectors create a checklist of their expectations at each milestone of the project.

This study was arranged to monitor training, designing, coding, and unit and system testing of a realistic industrial software development project. We initially planned to study the entire software development, but without a production-quality Ada compiler, the task was nearly impossible. However, many of our observations about early development may apply to other projects using Ada.

COMPUTER

1. Type of Error

- ☐ requirements incorrect
☐ requirements misinterpreted
☐ *design incorrect
☐ *design misinterpreted
☐ *code incorrect
☐ external environment misunderstood (not language or compiler)
☐ clerical error

*Was the error in the use of data _____ or in function _____?

2. Did the use of Ada as a design and implementation language contribute to this error? _____

If so, was it only a syntax error? _____

3. Whether related to Ada or not, which language features were involved in the error? _____

4. For an error in the PDL or code

- a. does the documentation explain the feature clearly? _____ Yes _____ No
 b. which of the following is most true?

- ☐ understood features separately, but not their interaction
☐ understood features but didn't apply them correctly
☐ didn't understand features fully
☐ confused feature with a feature in another language

c. where was the information needed to correct the error found?

- ☐ class notes
☐ Ada reference manual
☐ another programmer
☐ remembered
☐ viewgraphs from tapes
☐ test program
☐ other: _____

5.

	Detecting error:		Isolating source:	
	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Design reading				
Design walkthrough				
Code reading				
Code walkthrough				
Talk w/other programmer				
Reading documentation				
Compiler messages				
System error messages				
Project error messages				
Trace				
Dumps				
Inspection of output				
Pre-acceptance test run				
Acceptance test				
Ada runtime checking				
Other:				

6. What was the time used to isolate the source of the error?

1 hr. 4 hrs 2 days 1 wk. 2 wks. 1 mo. 2 mos.

If never found, was a workaround used? _____ (Explain in 8, below.)

7. When did the error enter the system?

- ☐ requirements ☐ design ☐ Ada coding ☐ testing
☐ implementing another change, Change Report No. _____
☐ other or can't tell (explain below)

8. Use this space to give any additional information that might help in understanding the cause of the change and its ramifications:

Programmers and training. Four programmers with diverse backgrounds were selected in order to examine whether a programmer's experience and education would influence his understanding and use of Ada. Table 1 shows the education, experience, and language knowledge of each programmer, but a sample of four programmers can only hint at possible influences. A more detailed study of background and performance is presented by Bailey.⁵

None of the programmers knew Ada before the project began; they volunteered to learn Ada. Therefore, they were probably more enthusiastic than most programmers about using the language. Since this was an early project, the training was longer (one month) and more comprehensive than the industry standard at the time.

The training began with 15 hours of videotaped lectures by Ichbiah, Firth, and Barnes over a period of four days. Six days of in-house training by George W. Cherry in Ada syntax and basic concepts spread over the following four weeks. During this time, the programmers also practiced writing Ada programs, read the Ada reference manual, and reviewed their class notes. The NYU Ada/Ed interpreter was used for programming assignments, which included a 500-line team project.

However, as usual, the programmers had little experience with many of the software engineering practices that Ada was designed to support.⁶ They did have varying degrees of experience with structured software development practices, e.g., design and code walkthroughs, structured programming, and program design language. They were given a half-day review of software development practices by Victor R. Basili to provide them with a common perspective on these techniques. At the time, such training in methodology was considered sufficient; however, further discussions on method-

Figure 2. Error report form.

Table 1. Backgrounds of programmers.

PROGRAMMER	YEARS OF PROFESSIONAL EXPERIENCE	EDUCATION	LANGUAGES KNOWN
LEAD	9	B S. (COMP SCI.)	FORTRAN, ASSEMBLER
SENIOR	7	M. S. (COMP SCI.)	FORTRAN, ASSEMBLER, SNOBOL, PL/1, LISP
JUNIOR	0	B S. (COMP SCI.)	FORTRAN, ASSEMBLER, PASCAL, PL/1, LISP
LIBRARIAN	0	HIGH SCHOOL DEGREE	FORTRAN

Table 2. Size characteristics of the product.

SIZE MEASURE	PROGRAMMER				
	LEAD	SENIOR	JUNIOR	LIBRARIAN	TOTAL
PDL					
NON-BLANK LINES	1364	2301	1430	203	5298
TEXT LINES	560	978	891	117	2546
EXECUTABLE STATEMENTS	266	421	343	65	1095
COMPILATION UNITS	9	14	19	1	43
ADA					
NON-BLANK LINES	1247	3611	3509	145	8512
TEXT LINES	706	1904	1648	117	4375
EXECUTABLE STATEMENTS	290	718	661	62	1731
COMPILATION UNITS	4	20	24	1	49
TOTAL (ADA AND NON-EXPANDED PDL)					
NON-BLANK LINES	1633	3611	4307	396	9899
TEXT LINES	857	1904	2159	274	5154
EXECUTABLE STATEMENTS	378	718	866	127	2089
COMPILATION UNITS	9	20	36	2	67

ology, especially the use of abstractions during design, occurred during the project as the programmers used the techniques.

This approach (Ada first, software engineering techniques later) did not seem to give the programmers the appropriate model for learning how to use Ada to support the software engineering concepts. They had only the programming model from their previous language experience, predominantly Fortran. Prior training in the software engineering concepts might have better prepared them for learning Ada. Bailey attempted to compare the two training approaches (Ada first and software engineering concepts second versus the opposite ordering).⁵ His study tried to correlate the programmers' background and the order of concept presentation with success in the classes he studied. However, the results were inconclusive and indicated that more studies are needed to determine how Ada should be taught.

Development and product. The project under study involved the re-design and implementation in Ada of a portion of a satellite ground control system originally written in Fortran. It included an interactive operator interface, graphic output routines, and concurrent telemetry monitoring. The programmers never saw the comparable Fortran source programs. Because pieces of the subset were scattered throughout the original system and the developed system contained some added features, determining the precise size of the subset as implemented in Fortran was difficult. However, the subset was estimated to contain between 5000 and 8000 text lines of Fortran, including declarations but not comments or blank lines.

The project began in February 1982 and ended in July 1983. However, most of the development took place between February and December 1982. Some testing was done between May and July 1983. Requirements

analysis was done prior to and concurrently with training, and an Ada-like Program Design Language was used to design the system.

The PDL has two levels. The first describes the input, output, and exceptions for the module, includes a brief abstract of what the module should do, and outlines the algorithm. The entire first level is written in Ada comments.

The second level is a more detailed description of the algorithm with a combination of Ada and PDL escapes enclosed in braces. These escapes can replace any Ada nonterminal, though they usually replace statements and conditionals. The first level remains as documentation for the code.

After design, the system was coded in Ada, and some of it was unit tested. As the project was begun before production-quality Ada compilers were available, it was not completely coded or tested. About 750 lines of PDL text were left uncoded. Some unit testing was done with the NYU Ada/Ed interpreter and the ROLM compiler, but no system testing was conducted.

The product was examined by a number of people interested in but not associated with the project, such as the developers of the original project. The design was judged to be functional rather than object-oriented. This assessment was not surprising since the programmers were most familiar with Fortran and its functional approach and the requirements were functional in nature. In fact, the high-level Ada design was very similar to the original Fortran design.

In order to provide an initial characterization of this project, Table 2 provides some size data for the design and code. Note that some design was never expanded into code because the project was not completed. In addition, many sections of code were copied almost verbatim from the corresponding PDL. All but four of the modules with both PDL

and code had a text expansion ratio below two to one. Of the remaining four modules, three had expansion ratios just over two to one, and one module expanded from 29 text lines to 124. Therefore, the total section includes only code and non-expanded design. Nonblank lines of source include comment lines but not blank lines. Text lines must have some Ada or PDL on them. Executable statements do not include declarations unless there is an initialization in the declaration.

Factors affecting the data

Several factors affected the outcome of this study, and understanding them is important for proper interpretation of the results. Many will not be present in later Ada developments, but the training and tool issues that clearly affected this project will affect others as Ada use increases.

The useful, but very slow, NYU Ada/Ed interpreter became unusable toward the end of the project, as the size of the developing system grew. This difficulty had a demoralizing effect on the programmers, and they did not finish coding or testing the project. When the ROLM compiler became available, further testing was done.

The results set forth are based on data collected through coding and some unit testing. In addition, the vast majority of the Ada-related errors either were or could have been detected by a compiler. The dominance of these errors might have diminished had the code been executed and testing completed. Many more logic errors might have been uncovered had all the modules undergone error-free compilation.

Many trivial errors that might have been detected by a PDL processor appeared in the design. Some of these were detected during design readings and reviews and were removed. Others

Table 3. Effort for each phase of the project.

PROJECT PHASE	AMOUNT OF TIME (IN HOURS)	PERCENTAGE
REQUIREMENTS ANALYSIS	530.5	12.73
REQUIREMENTS WRITING	113.6	2.73
DESIGN CREATION	514.4	12.34
DESIGN READING	37.7	0.91
FORMAL DESIGN REVIEW	162.4	3.89
CODING	305.6	7.33
CODE READING	13.3	0.32
FORMAL CODING REVIEW	62.3	1.50
UNIT TESTING	332.7	7.98
INTEGRATION TESTING	0	0.00
REVIEW TESTING	0	0.00
TRAINING	849.1	20.38
OTHER ACTIVITY	1245.7	29.89
TOTAL REQUIREMENTS	644.1	15.46
TOTAL DESIGN	714.5	17.14
TOTAL CODE DEVELOPMENT	381.2	9.15
TOTAL TESTING	332.7	7.98
TOTAL TRAINING	849.1	20.38
TOTAL OTHER ACTIVITY	1245.7	29.89
ENTIRE PROJECT	4167.3	100.00

remained until the code developed from the design was compiled. Many of these later changes were not made in the design; therefore, some of the design and code documents were inconsistent. In addition, design reviews tended to focus on the numerous, easily detected, trivial errors rather than on the deeper design issues and, perhaps, errors. A PDL processor would have changed this focus.

Type and quantity of training were other factors. Twenty percent of the total effort was spent on training. Software engineering concepts such as data abstraction and information hiding were not stressed during Ada training, although they were presented to the programmers afterward. The programmers indicated that training was insufficient, and their use of Ada suggests that they probably needed more. Therefore, we must conclude that a sizable effort will be needed to learn Ada and must be considered when planning early projects using the language.

Effort

The first goal of the study is to characterize the effort expended on the project. By doing so, we can provide insight into how programmer time might be used in future Ada projects and a basis for comparison with later Ada projects.

Table 3 shows the time spent on each phase of the project, including training. Productivity was calculated from the total lines in Table 2 and the total design and code development time in Table 1. For each hour spent in design and code development, 9.03 nonblank lines of code and 4.70 text lines were developed. The values are upper bounds (and may not be meaningful since the project was not completed).

Changes

Our second goal is to characterize the changes in the project in order to

Table 4. Breakdown of changes by type.

TYPE OF CHANGE	NUMBER OF CHANGES	PERCENTAGE
ERROR CORRECTIONS	192	56.96
CHANGES IN PROBLEM DOMAIN	1	0.29
PLANNED ENHANCEMENTS	9	2.67
AVOIDANCES OF APPARENT PROBLEMS WITH THE COMPILER	18	5.37
AVOIDANCES OF OTHER PROBLEMS IN THE DEVELOPING ENVIRONMENT	2	0.59
ADAPTATIONS TO A CHANGE IN THE DEVELOPING ENVIRONMENT	7	2.08
IMPROVEMENTS OF DOCUMENTATION, CLARITY, OR MAINTAINABILITY	76	22.55
OPTIMIZATION OF TIME, SPACE, OR ACCURACY	2	0.59
INSERTION OR DELETION OF DEBUG CODE	9	2.67
OTHER THAN ABOVE	21	6.23

determine how the product evolves. The classification of changes can indicate which factors might have affected the project. Information on how easily the product was changed might indicate the quality of the product.

Analysis of the 337 change request forms (Figure 1) and the 439 individual document change forms indicates that the effect of Ada on the changes made in the project cannot be distinguished from the effect of any other factor. Code changes accounted for 61 percent. As stated previously, however, many of these changes were errors which should have been caught at the design stage. Thirty-two percent of the changes were in design documents, and only seven percent were in requirements documents.

The breakdown by type of change is shown in Table 4. The majority (57 percent) of the changes were error corrections which will be described in detail later. Of the non-error changes, 52 percent were improvements of clarity, maintainability, and documentation. The low number of planned enhancements indicates that the programmers tried to implement portions of the system immediately rather than start with a subset and enhance it later. The large number of improvements

show that they were concerned about clarity and documentation.

The time to determine the need for change was one hour or less in almost all cases. In addition, 46 percent required only six minutes. The need for these changes was easily determined. Few changes took much longer than a half hour, although four changes required more than one day to determine they were needed: two were planned enhancements; one was an avoidance of a problem with the compiler; and the last involved the creation of a global definitions package that interfaced with several components.

The amount of time needed to design and implement changes was also minimal. The majority took one hour or less. Of the code changes, all but five took two hours or less. Two changes, which took three hours and one day, respectively, involved avoiding problems with the compiler. One change, which took one and a half days, was an adaptation to a change in the development environment. One code change, which took four hours, was an error correction and will be discussed in the errors section. The global definitions package was implemented in four days. The few other changes which took much longer than

usual were mostly planned enhancements and improvements of clarity, maintainability, and documentation of requirements documents. A change that took one week was a planned enhancement in a requirements section.

The total time spent determining the need for changes then implementing them was 426.4 hours, 10 percent of the total effort for the entire project. The average cost was 1.27 hours per change, but more than 80 percent of the changes took much less time.

Components involved. We determined the number of components altered in each change. Seventy-seven percent of the changes caused only one component to be modified, but up to five components were modified in some changes. We also identified 70 interface changes (21 percent of all changes) defined as those that entail a change in more than one component at the same level of document. Only 2.9 percent of these were in the requirements; the rest were equally divided between design and code. As many as five components were altered in these interface changes.

From this data, we conclude that most of the changes were trivial and involved a single component. Ada seemed to have little effect on the non-error changes. Most of the changes were error corrections, but many were improvements of documentation, clarity, or maintainability. We do not know how this distribution would change if more testing were done; but, we strongly suspect that the number of error corrections would increase.

Errors

Since Ada is a new language, programmers will make some errors when using its new features. By determining types of errors made, we can focus training, tools, and techniques on eliminating or detecting the most prevalent or severe errors.

We examined 192 error description forms (Figure 2). Each corresponds to a change request that falls into the error correction category. We used several different error classification schemes to understand which errors occur and how to detect or prevent them. Note that our figures (Table 5) differ slightly from Basili and Pericone³ because some classifications were changed, and the data were interpreted in light of these changes.

We used the definitions of errors, faults, and failures of the IEEE Glossary of Software Engineering.⁷ A "fault" is a specific manifestation in the source code of a programmer "error." A single "error" can result in many "faults." A "fault" may cause a "failure" when the program is executed. Errors were reported for this project, but few, if any, failures were reported because little testing was done.

Document type. A common classification of errors is by type of document and how it was involved. If we know the documents involved, we can examine them more closely for faults or concentrate on their careful development. Table 5 shows a breakdown of the errors by document type. We can see that the majority (79 percent) of errors were due to incorrect code. Most of the remaining errors were attributable to incorrect design. Few errors involved those requirements, probably because those requirements had already been used on a previous project and were fairly well written.

Detection and correction. If we knew which activities were most often successful at detecting errors, we could concentrate training and tool development to support them. In this project, compilation, design reading, design walkthroughs, and code reading were most often used to detect errors. Approximately half of the errors

Table 5. Errors by type of document.

TYPE OF DOCUMENT AND HOW INVOLVED	NUMBER OF ERRORS	PERCENTAGE
REQUIREMENTS INCORRECT	2	1.04
REQUIREMENTS MISINTERPRETED	4	2.08
DESIGN INCORRECT	29	15.10
DESIGN MISINTERPRETED	0	0.00
CODE INCORRECT	151	78.65
EXTERNAL ENVIRONMENT MISUNDERSTOOD (NOT LANGUAGE OR COMPILER)	0	0.00
CLERICAL ERROR	6	3.12

were successfully detected through compiler messages, and a slightly smaller number were successfully detected through readings and walkthroughs. These same activities were used to isolate the source of the error. Code reading was more successful at isolating than detecting the source, and the opposite is true of compiler messages. In the case of design reading and walkthroughs, detection of the errors and isolation of their sources usually occurred simultaneously. This information indicates that careful design and code reading and walkthroughs should be stressed and that language processors should be used as much as possible to detect errors. However, results of other activities, such as test runs, would surely have appeared here if more testing had been done.

As with the changes, most of the errors were trivial. More than 80 percent took 12 minutes at most to isolate. Only seven errors took an hour or more to either isolate or to correct. One error—a design incorrect error that involved renaming a file—took an hour to isolate but only six minutes to correct. Another, classified as code incorrect, took two hours to isolate but only twenty minutes to correct. An undefined part of a string was passed as an argument to a function. Two errors involving incorrect design each required only six minutes to isolate but more than an hour to correct. One of these, a tasking error involving a syn-

chronization problem between two components, took 5.2 hours. Another, which required 1.5 hours, was a logic error involving input/output. The remaining three errors took an hour or more to isolate and another hour or more to correct. One required the insertion of error checks and exception handlers in a routine conforming to the specifications; this took one hour to isolate and one hour to correct. Another took four hours to isolate and four hours to correct; it was an input/output syntax error. The last error, which took one hour to isolate and one hour to correct, was a requirements incorrect error. A superfluous requirements section was found and eventually deleted.

Table 6 shows the number of components changed to correct each error as well as the number examined while deciding how to make the correction. (A distinction is made between errors in general and those which caused compiler-detectable faults. This distinction will be described in more detail in the next section.) Since most of the errors were trivial and involved the syntax of a component, most of the corrections caused only one component to be changed or examined.

Possible detection by tools. Detection by tools is one method of classifying faults. The classification will be expanded in later studies, but we used it here to separate compiler-detectable from noncompiler-detectable faults.

Table 6. Number of components involved in error correction (with all errors and without compiler detectable faults).

NUMBER OF MODULES INVOLVED	ALL ERRORS		W/O COMPILER FAULTS	
	CHANGED	EXAMINED	CHANGED	EXAMINED
1	173	167	35	31
2	16	20	7	9
3	3	5	2	4

Table 7. Errors classified by which tool would detect them.

WHICH TOOL WOULD DETECT	NUMBER OF ERRORS	PERCENT OF TOTAL ERRORS
Compiler	148	77
BNF	80	42
Not BNF	68	35
Not Compiler	44	23

Table 8a. Number of reported errors in module.

# ERRORS IN MODULE	# MODULES WITH ERRORS		TOTAL	
	ADA	PDL	ADA	PDL
0	SSSJJJJJJ	LLLLLLLLJJ	11	13
1	LSSJJJJ	LLL	8	3
2	LSJ	L	3	1
3	SSJ		3	0
4	SSSSJJJB	B	8	1
5	SSJJ		5	0
6	LSSJ		4	0
7	SJ		2	0
> 10	LSSJJ		5	0

Table 8b. Number of reported errors in module (without faults detectable by compiler).

# ERRORS IN MODULE	# MODULES WITH ERRORS		TOTAL	
	ADA	PDL	ADA	PDL
0	LLSSSSSSJJJJJJJJJJJJJJ	LLLLJJJJJJJJJJJJ	25	15
1	LLSSSSSSJJJJJJJB	L	15	1
2	SSJJ	LB	4	2
3	SS		2	0
4	S		1	0
5	S		1	0
10	S		1	0

L: Lead programmer S: Senior programmer J: Junior programmer B: Librarian

The compiler-detectable faults are further divided into those related to the BNF of the language and those that might require more information than the BNF contains. Those faults detect-

able by a processor based on BNF should be eliminated with a syntax-oriented editor, which might also eliminate some of the other faults. Compiler-detectable faults have often

been removed by the time data collection begins on many projects. Therefore, the data from which those faults have been removed might be comparable to early development data in later projects. Table 7 lists the data for this project using this classification scheme.

Number of errors per module.

Tables 8a and b depict how many errors were reported in each of the 67 modules (Ada and non-expanded PDL). Table 8a shows the total errors reported, and 8b itemizes the number of errors reported in which the fault was not detectable by a compiler. The letters in each row indicate which programmer wrote the module. The modules with more than ten errors had 15, 11, 20, 12, and 27 reported errors, respectively.

Further processing after project completion showed that most of the non-expanded PDL modules had compiler-detectable faults even though no errors in those modules were reported. This finding indicates that the modules were written, expanded into code, then essentially ignored. The data in Tables 8a and 8b reinforce this observation. The senior programmer seems to have found more of the less obvious errors than other programmers since his modules have more reported errors that were not compiler-detectable.

Omission or commission. Another type of error classification, presented by Basili and Perricone,⁸ divides errors into the categories of omission and commission. Errors of omission leave out some portion of code while errors of commission include erroneous or superfluous code. Table 9 presents the data for this project as well as some of the data from Basili and Perricone. Note that the percentages for all errors from this study and in new modules from the earlier study are almost the same. This is probably coincidence, since the data was gathered at different times during develop-

Table 9. Comparison of errors of omission and commission.

ERRORS INVOLVED	RAW ERRORS		PERCENTAGE	
	OMISSION	COMMISSION	OMISSION	COMMISSION
This study				
All errors	89	103	46	54
w/o compiler faults	23	21	52	48
Basili & Perricone				
All errors	79	143	36	64
New module errors	52	63	45	55

ment, and our data are incomplete. This categorization will be included in some of the following tables. Errors of omission generally will not be caught by testing with a structural coverage criterion and may be overlooked in code reading.

Language, problem, or clerical. We developed yet another classification scheme where the errors are identified as language, problem, or clerical. Language errors are closely related to the use of Ada and are further classified as concept, semantics, or syntax. A syntax error involves a misunderstanding or misuse of the syntax of a feature; a semantics error involves a misunderstanding of the meaning of a feature in that language; and a concept error involves a misunderstanding of a feature's use. The problem category results from a misconception of the problem domain or the environment. Clerical errors include those due to carelessness, e.g., typographical errors. This classification is somewhat subjective, however, since the project monitors tried to determine what the programmer was thinking when the error occurred.

Of the 192 error description forms examined, 160 (83 percent) claimed that the use of Ada contributed to the error. As shown in Table 10, the majority of the errors were language errors, and 67 percent of those were syntax errors, which explains why so many of the errors took so little time to correct. Almost 21 syntax errors oc-

Table 10. Number of language, problem, and clerical errors.

CATEGORY	NUMBER OF ERRORS	
	ALL ERRORS	W/O COMPILER FAULTS
Language	160	18
Concept	8	8
Semantics	44	10
Syntax	108	0
Problem	26	26
Clerical	6	0

curred per thousand lines of text (any line containing part of an Ada statement) and almost 11 syntax errors per thousand non-blank lines.

The language-problem-clerical classification can be used in conjunction with the document-type classification as seen in Table 11. Not surprisingly, most errors involving requirements were problem errors, and most of the errors involving incorrect design or code were language-related errors.

Ada language features. Several Ada language features were involved in errors. Understanding the relationships between errors and features may help prevent the errors. Table 12 shows the language features involved in errors, with all reported errors included. In Table 13, the errors that caused compiler-detectable faults have been removed.

Low-level syntax (e.g., semicolon, parenthesis, assignment), loops, declarations, and parameters were involved in the most common language errors.

Several errors also involved tasks, separate compilation, generics, and procedures and functions. As previously stated, most of the errors were compiler-detectable. Only eight concept errors, which involved tasking, exceptions, and packages, occurred. Of the 44 semantics errors, nine involved parameters; six, generics; five, compilation units; four, declarations; and three, overloading. However, many of those could be detected by a compiler. If the compiler-detectable faults are removed, only 10 semantics errors remain, and four of those involve parameters.

In general, few serious errors were reported in this project because little testing was done. However, the data reported suggests the types of errors to expect when people given training similar to that of our programmers learn to use Ada. Similar errors might be made when learning any new language, however, particularly with such a large number of new features and concepts.

Table 11. Type of document vs. language, problem, or clerical classification and omission or commission classification (data excluding compiler detectable faults in ()).

TYPE OF DOCUMENT AND HOW INVOLVED	LANG	NUMBER OF ERRORS			
		PROB	CLER	OMISSION	COMMISSION
Requirements incorrect	0	2(2)	0	1(1)	1(1)
Requirements misinterpreted	1(1)	3(3)	0	3(3)	1(1)
Design incorrect	24(8)	5(5)	0	14(6)	15(7)
Design misinterpreted	0	0	0	0	0
Code incorrect	135(9)	16(16)	0	70(13)	81(12)
External environment misunderstood	0	0	0	0	0
Clerical error	0	0	6(0)	1(0)	5(0)
Total	160(18)	26(26)	6(0)	89(23)	103(21)

Table 12. Errors categorized by Ada language feature.

ADA LANGUAGE FEATURE	NUMBER OF ERRORS					TOTAL
	CON	SEM	SYN	OM	COM	
Semicolon	0	0	17	13	4	17
Parenthesis	0	0	12	9	3	12
Colon	0	0	4	2	2	4
Assignment	0	0	5	1	4	5
Strings	0	0	4	3	1	4
Comment	0	0	4	2	2	4
Identifier	0	2	3	0	5	5
Loop	0	2	8	7	3	10
Case	0	0	1	1	0	1
If	0	0	6	2	4	6
Begin/end	0	0	4	3	1	4
Return	0	0	1	0	1	1
Scoping	0	0	2	1	1	2
Typing	0	2	5	0	7	7
Aggregate	0	1	0	0	1	1
Arrays	0	2	2	1	3	4
Records	0	2	2	2	2	4
Declarations	0	4	8	7	5	12
Parameters	0	9	5	3	11	14
Procedures & functions	0	2	5	2	5	7
Access type	0	1	0	0	1	1
Tasking	5	0	4	4	5	9
Exceptions	2	0	1	0	3	3
Generics	0	6	2	3	5	8
Packages	1	0	1	2	0	2
Compilation units	0	5	2	5	2	7
Attributes	0	1	0	0	1	1
Pragmas	0	2	0	0	2	2
Overloading	0	3	0	0	3	3
Totals	8	44	108	73	87	160

Ada use

A description of how the language is used is the fourth goal. Since Ada is complex, it was thought that the pro-

grammers might begin with a subset of the language. Ada also supports a number of software engineering concepts such as information hiding and abstraction. Assessing Ada use might

aid in the evaluation and modification of training in Ada concepts and applications. In addition, tools might be developed to help people learn to use Ada's more unusual features.

By examining its simplest features, we discovered that except for the goto and code statements and representation clauses, the programmers used all of Ada's syntactic features. Tasking, generics, packages, exceptions, and overloading along with pragmas, aborts, and delays were used nominally. However, when the system was designed, the programmers did not know how to use these concepts on this application. Therefore, they might have been uncomfortable basing their design on some of Ada's more advanced features. If the programmers had more examples within their application domain, they might be able to take advantage of these features.

We also looked at the use of packages in the system to determine whether concepts such as data encapsulation and information hiding were used effectively. The senior and junior programmers defined 11 packages for use with this project; however, the lead programmer and librarian defined none. Two of those 11 packages served as definition common blocks; three were libraries of functions; four defined encapsulated data types exporting private-type definitions and operations; and the remaining two defined types but exported the representation of the type. Of the four packages which defined encapsulated data types, two were device drivers and one was a mathematical function; the remaining package definition had no corresponding body. This indicates that no new encapsulated data types were defined. The programmers used packages for types they had used in other languages. While globally visible, many of these packages were not needed globally, indicating that the programmers did not understand the concept of information hiding. Can-

non describes the use of packages in greater detail.²

Most programmers are not accustomed to high-level language support for their concurrency needs. Familiarity with concurrency in another language would be of little benefit, however, as Ada uses an unusual model, rendezvous and tasks, for concurrency. We wanted to know how tasks were used in this system. Although the system was designed with communicating tasks, they were at a high level and had little communication. Ten tasks were defined: one by the lead programmer, four by the senior programmer, and five by the junior programmer. Except for two cases, each task had only one or two entries, and since the system was not tested, it is difficult to know whether this use was appropriate or, indeed, if it worked. Only further experience with tasks would determine their use. Training in tasks, like packages, should probably include examples from the appropriate application domain.

We also sought an understanding of the exceptions used. However, it remains unclear when exceptions should be handled in the module raising them and when they should be propagated. Nevertheless, the programmers tried to use exceptions, if only for passing back error codes. Twenty-one of the non-package modules had exception handlers, and exceptions were raised explicitly in 17 modules. Without knowing whether the system runs, it is difficult to ascertain whether this use of exceptions is sufficient or appropriate.

The results of this portion of the study are mixed. While the programmers used many features of the language, it is difficult to determine whether that use was nominal or appropriate. Furthermore, we know little about how Ada should be used. Most examples in the literature are too small to compare with this project. How-

Table 13. Errors categorized by Ada language feature (without compiler detectable faults).

ADA LANGUAGE FEATURE	NUMBER OF ERRORS				TOTAL
	CONCEPT	SEMANTICS	OMISSION	COMMISSION	
Loop	0	2	2	0	2
Arrays	0	1	0	1	1
Parameters	0	4	0	4	4
Procedures & functions	0	1	1	0	1
Tasking	5	0	2	3	5
Exceptions	2	0	0	2	2
Generics	0	2	2	0	2
Packages	1	0	1	0	1
Totals	8	10	8	10	18

ever, no discernible subset was defined. Other than the code statements and representation clauses, which were not needed for this application, most of the language was used. Therefore, use of a "subset compiler" would not have been appropriate and might have limited the programmers' design of the system.

Programmer differences

The fifth goal discussed includes a description and evaluation of the differences between programmers and their use of Ada. Programmers with varied backgrounds might use Ada differently and might make different types of errors. If these differences are significant, they might suggest differences to be seen in other environments. They might also suggest how to tailor training to meet the needs of programmers with varied backgrounds.

We found that the programmers used most of the language in basically the same way. The librarian wrote so little code that drawing any conclusions about that code or programmer would be presumptuous. Other than their definition and use of packages, the other programmers' code is basically indistinguishable. Either we do not have the appropriate techniques

for examining their code or they worked so closely together that their individual differences are hidden.

Productivity is one area where some differences between programmers surfaced. While the rest of the programming team produced 7.3 lines of code per hour spent in design and code development, the senior programmer produced 16.5 lines per hour. By all reasonable measures of productivity, the senior programmer was most productive. The junior programmer was somewhat more productive than the lead programmer and the librarian. The fact that the junior and senior programmers wrote the most code and became most familiar with Ada may explain the disparities in performance.

The only marked difference among programmers' errors was that the junior programmer made the most language, and particularly syntax, errors. However, he performed the most code testing and therefore had the greatest opportunity to discover errors. He also had the most extensive background in software engineering methodology, which seemed to help him understand how to use Ada and offset his lack of experience in the application area.

Overall, the programmers seemed to write code using the features of the

language they thought they knew best. The senior and junior programmers, who had varied language experience, used the more Ada-like features such as packages, but the lead programmer also used tasking and generics. The librarian, with little language experience, used a simple subset of Ada. He only wrote two modules, which required only a subset of Ada. None of the programmers made errors remarkably different from those of the others, although further testing might have shown otherwise. Productivity appears to be the only aspect of this project that could be used to differentiate programmers.

Although the project ended before development was complete, the results indicate what might happen in early stages of development in other projects. A number of results from this project might prevent others from making costly management mistakes.

Above all, it should be noted that learning Ada takes time, a factor that will influence any estimate of effort for early projects using Ada. Training will probably have to continue as team members learn the finer points of the language.

Ada is more than syntax and simple examples. The underlying software engineering concepts must be taught in conjunction with the support Ada provides for those concepts. Examples from the relevant problem domain will help students fit Ada into their environment. Since most programmers are not familiar with the methodologies developed in the 70's, which Ada supports, training in software engineering methodology and its use in the environment of a particular application is an absolute necessity.

How Ada should be used remains unclear. Ideally, our understanding of the software engineering concepts that Ada supports would simplify its use. However, many people learn by example, and good examples are lacking.

We neither know how nor when to use exceptions, tasks, and generics, and can only gain this knowledge by studying various alternatives and showing how they work with examples from various environments. In this respect, the project has raised more questions than it has answered.

Design alternatives must be investigated. The design for this project was functional and more like than unlike the earlier Fortran design. A group at General Electric developed an object-oriented design for the same project,⁹ and it is not clear which design, if either, is most appropriate. Just as a combination of top-down and bottom-up development is appropriate to many applications, a combination of functional and object-oriented design might well be most appropriate. Only by determining which design type, or combination of types, is best suited to the particular application can we teach people which design approach to use. Without such training, programmers must rely on their experience with other languages and will probably produce functional designs.

Proper tool support is mandatory. This project was undertaken without a production-quality validated compiler—a necessary tool. Likewise, a language-oriented editor, capable of eliminating 60 percent of the observed errors, would have been desirable. Such an editor would have freed the programmers to concentrate on the logic errors that undoubtedly remain in the design and code. Such an editor would have dramatically reduced the error rate. Other useful tools for this project would be data dictionaries, call structure and compilation dependency tools, cross references, and other means of obtaining multiple views of the system. A PDL processor with interface checks, definition and use relation lists, and various metrics would also have aided in the early stages of development.

Some methodology must be fol-

lowed for a project to be successful, and programmers must understand the methodology and tools before the project begins. In this case, the lack of useful tools proved troublesome. In addition, the PDL was loosely defined until after design began. Effective design reading might have caught many errors. If we had tested this project after a compiler became available, a test plan created after the requirements were completed would have been necessary. However, that aspect of the methodology was deemed unimportant. Language is only one aspect of the environment and methodology. It cannot save a project in which the rest of the methodology is ignored.

We believe this project is atypical since it was not finished and no compiler was available. However, it is typical in that training consumed an enormous amount of effort, and the programmers were not familiar with the underlying software engineering concepts of Ada. In this respect, it resembled the beginning of many projects. Also of note, the learning curve in methodology is quite large. As we study more projects that use Ada, we will learn how to both teach and use it and discover how to reduce mistakes. In the meantime, we know that using Ada will be difficult at first, but in time its use will make us more effective in applying existing software engineering techniques to ease the programming process and thereby increase the quality of the product. □

Acknowledgments

Elizabeth Kruesi Bailey, John W. Bailey, John D. Gannon, Sylvia B. Sheppard, and Marvin V. Zelkowitz were the other monitors of this project and contributed to the work reported here. The authors would also like to thank the other members of our research group, particularly David H.

Hutchens, James T. Ramsey, and Richard W. Selby, Jr., for numerous enlightening discussions concerning this project. Research for this study was supported in part by the Office of Naval Research and the Ada Joint Program Office under grant N00014-82-K-0225 to the University of Maryland.



Victor R. Basili is professor and chairman of the Computer Science Department at the University of Maryland. He is currently measuring and evaluating software development in industrial settings.

Basili was involved in the design and development of several software projects, including the SIMPL family of programming languages. He has authored over 60 published papers. In 1982, he received the Outstanding Paper Award from the *IEEE Transactions on Software Engineering*. He was program chairman for both the Sixth International Conference on Software Engineering and the First ACM SIGSOFT Software Engineering Symposium on Tools and Methodology Evaluation, and serves on the editorial boards of the *Journal of Systems and Software* and the *IEEE Transactions on Software Engineering*. He is a member of the ACM and the executive committee of the Technical Committee on Software Engineering, and is a senior member of the IEEE CS.



Elizabeth E. Katz's research interests include the measurement and evaluation of the effect of various tools and techniques on the software development process and its product, as well as the development of such tools and techniques. Her current focus is on measures for developments using Ada. She received a BS degree in computer science and English from the College of William and Mary in 1981, an MS in computer science from the University of Maryland in 1983, and is now working toward her PhD at Maryland. She is a student member of ACM and IEEE.

Questions about this article can be directed to Victor R. Basili, Dept. of Computer Science, University of Maryland, College Park, MD 20742.



Nora Monina Panlilio-Yap is a research assistant at the University of Maryland. Her area of research is software engineering. She obtained the BS in chemical engineering from the University of the Philippines in 1976, and the MA in computer science from Duke University in 1982. She is currently working towards the PhD in computer science at the University of Maryland. She was a World Fellowship recipient of the Delta Kappa Gamma Society International from 1979 to 1984 and has been an International Fulbright Scholar since 1979.



Connie Loggia Ramsey is a doctoral candidate and research assistant in computer science at the University of Maryland, College Park. Her research interests include the development of expert systems for software engineering. She received her BA in biology from the State University of New York at Binghamton in 1979.



Shih Chang is interested in software development techniques and tools. He received a BS degree in computer science from the University of Maryland in 1983, where he is currently a graduate student in computer science. He is a student member of the ACM and IEEE Computer Society.

References

1. V. R. Basili and E. E. Katz, "Metrics of Interest in an Ada Development," *IEEE Workshop on Software Engineering Technology Transfer*, Miami, FL, Apr. 1983, pp. 22-29.
2. J. D. Gannon, E. E. Katz, and V. R. Basili, "Characterizing Ada Programs: Packages," *The Measurement of Computer Software Performance*, Los Alamos National Laboratory, Aug. 1983.
3. V. R. Basili et al., "A Quantitative Analysis of a Software Development in Ada," University of Maryland tech. report UOM-1403, 1984.
4. V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering*, Vol. SE-10, No. 6, Nov. 1984, pp. 728-738.
5. J. W. Bailey, "Teaching Ada: A Comparison of Two Approaches," *First Washington Symposium on Ada Acquisition Management*, ACM, Laurel, MD, March 6, 1984.
6. M. V. Zelkowitz et al., "Software Engineering Practices in the US and Japan," *Computer*, Vol. 17, No. 6, June 1984, pp. 57-66.
7. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE-STD-729-1983, IEEE, New York, 1983.
8. V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Vol. 27, No. 1, Jan. 1984, pp. 42-52.
9. A. G. Duncan et al., "Communications System Design Using Ada," *Proc. Seventh Int'l. Conf. Software Engineering*, 1984, pp. 398-407.

END
DTIC
FILMED
4-86